The Program Composition Project

K. Mani Chandy and Stephen Taylor
Carl Kesselman
Ian Foster

Computer Science Department
California Institute of Technology

Caltech-CS-TR-90-03

The Program Composition Project

K. Mani Chandy and Stephen Taylor
California Institute of Technology

Carl Kesselman
The Aerospace Corporation

Ian Foster
Argonne National Laboratory

February 4, 1990

Caltech-CS-TR-90-03

1 Introduction

The Program Composition Project is a research effort that aims to improve the usefulness of parallel architectures by providing innovative programming concepts based on formal theory. In essence, our vision of the future can be summarized as follows:

> *Parallel computing will be elevated to the point where it becomes the commonplace method of computing in the next century; parallel programming methods will replace conventional techniques at every level from the high-school classroom to the research laboratory.*

Sequential machines will increasingly become less competitive with parallel architectures from a cost-performance viewpoint and will eventually cease to be a realistic basis for computing.

Our central aim is to provide a *formal and practical foundation* for parallel programming. We plan to carry out two activities in order to demonstrate and evaluate this foundation:

- Building a portable, parallel-program development environment, and
- Implementing a variety of realistic, non-trivial applications.

We seek to promote the use of parallel machines by providing scientists with support for two fundamental activities: *building abstractions* and *experimenting with physical models.* The first of these activities is concerned with mastering program complexity; it will be supported via the Program Composition Notation (PCN), associated program development methodologies and formal theory. Experimenting with models will be supported via a collection of *workbenches.* A workbench comprises a set of generic programs oriented toward a specific application domain such as DNA sequencing or particle flow simulation.

This document provides an overview of the project; it describes the ideas involved in each of the core programming tools provided by the programming environment and describes the application efforts to be undertaken. A schedule of the activities for the first year is given in Appendix A.

1.1 Architectural View

Abstractly, a parallel architecture can be viewed as a connected graph of computing sites. From a programming perspective, we seek the ability to treat all computers in this graph as equivalent at some level of abstraction; thus, a program may execute on any site and may interact with programs executing at any other site. This view will be supported even if the sites are as dissimilar as an Intel i860 node, a Sequent Symmetry processor or a Sun Workstation. The symmetry of the system need only be broken at

the point of interaction with a scientist or programmer. The essence of a computer lies in the services that it provides to the programmer; we view the differences between machines largely as a parameter that can be gainfully employed for the purpose of optimization.

Given this view, the programming environment will be built in a portable manner that will allow a large class of computers to be employed in a parallel computation. Initially, this class will include Sun workstations, BBN Butterfly, Sequent Symmetry multiprocessors, and a variety of multicomputers including the Intel iPSC 2, Intel Delta and Symult s2010. We require only that terminal interaction be conducted via a UNIX-based machine.

1.2 Programming Approach

Our approach to programming parallel architectures is based on a single notion, *program composition*: combining programs, using well-defined methods, to form new programs. This concept has long formed the basis of program-development methodologies and research in formal semantics.

In programming conventional uniprocessors, languages have employed a single *sequential* form of composition; the advent of parallel machines has led to a variety of new methods by which to compose programs. It is our belief that the essence of parallel programming can be captured by *three* fundamental forms of program composition; these *composition operators* have been adopted in the Program Composition Notation (PCN)[2,3]. This notation, its formal theory and associated tools will form the project's programming foundation.

The principle program-development methodology used in the design of PCN programs is *stepwise refinement.* In this method, a program design begins with an initial specification and proceeds through a sequence of refinement steps. At each step, detail is added to the design so as to implement part of the specification. To aid the design process, correctness assertions may also be added to the program text and reasoning may be carried out to determine if the program is correct. Eventually, the refinement process terminates when elements of the specification correspond to elementary PCN statements.

A complementary development technique can also be employed to aid the design process: Existing program components, written in C, Fortran or Ada, may be used to implement part of a specification. This provides scientists with a method to utilize pre-existing algorithms and methods. In reasoning about the composite program, no attempt need be made to characterize the internal operation of existing program components. Instead, the components are characterized purely by their external interface stated in the form of correctness assertions; these assertions specify the preconditions and postconditions under which the components operate correctly.

Although PCN provides a set of fundamental methods by which programs can be composed, it also allows new forms of composition to be defined. These programmer-

defined compositions may be used to implement high-level abstractions specific to an application domain, such as fluid flow, weather analysis or DNA sequencing; these abstractions form the basis for *workbench* designs.

2 An Example Program

Figure 2.1 shows a PCN module that implements the well known, in situ quick-sort algorithm. The intention here is not to provide a complete description of PCN but rather to illustrate a number of fundamental concepts, namely, *definitions* and *state change* and *program composition.*

Definitions. Definitions can be viewed as variables that may go through only one state change: Initially they are *undefined*, eventually they are *defined* to be a value and subsequently they remain unchanged. Reasoning about parallel PCN programs is greatly simplified by virtue of this monotonic property. An equality operator (=) is used to make a definition; for example, in the qsort program, the definition s is defined to be the value at position lb in the array a (2).

State Change. Variables that occur in the declaration segment of a program are mutable: They initially contain an arbitrary value and may subsequently be modified many times. Mutable variables are used to represent program state that may change over time. Modifications to these variables are achieved using an assignment operator (:=); for example, in the qsort program, i is a mutable integer that is declared using an int declaration (1) and modified to hold the value of the definition *lb* (3). In contrast to the array a, the variable i does not occur in the argument list of the program; thus, it is allocated at program entry before the program body is executed.

Program Composition. PCN programs are composed using three methods: sequential (;), parallel (||) and choice (?) composition; the quick-sort program uses all three of these operators. Each has the form $\{ \mathsf{Op}\ \mathsf{P}_1, \mathsf{P}_2, \ldots, \mathsf{P}_n \}$, where Op is a composition operator and $\mathsf{P}_1 \ldots \mathsf{P}_n$ are the programs being composed.

Sequential composition is the most familiar and specifies that the component programs are executed in sequence; the composition terminates when P_n terminates. In parallel composition, programs are executed nondeterministically. The interleaving of programs is arbitrary but guarantees that an enabled program will eventually execute. A parallel composition terminates when all of its component programs have terminated. A program that appears in a parallel composition may be executed at another, remote computing site provided that all its arguments are definitions.

In choice composition, the programs composed are implications that have the form Guard $\rightarrow$ Body; the Guard specifies a sequence of preconditions (or tests) under which the Body program may be executed. A program whose Guard is satisfied is chosen nondeterministically; the Body is then executed and the composition terminates when the Body terminates. If no Guard is satisfied, the choice composition terminates.

It is possible to distinguish two dominant views of computation when programming

```
-exports(qsort)

qsort(lb,ub,a)                                  % quick sort array a
int a[],i,j,k                                   % declarations (1)
{ ? ub > lb →
    { ;   s = a[lb]                             % make definition (2)
          i := lb, j := ub, k := lb,            % make assignments (3)
          partition(lb,ub,i,j,s,a),             % partition the array
          swap(k,j,a),                          % put pivot in position
          {||   qsort(lb,j−1,a), qsort(j+1,ub,a) }   % sort parts in parallel
    }
}

partition(lb,ub,i,j,s,a)                        % partition array
int i,j,a[]
{ ?  i =< j →                                   % not done yet
    { ;   {||   movej(lb,j,s,a), movei(ub,i,s,a) },   % move in parallel
          i < j → swap(i,j,a),                  % intermediate point
          partition(lb,ub,i,j,s,a)              % continue
    }
}

swap(i,j,a)
int i,j,a[]
{ ;   tmp = a[i], a[i] := a[j], a[j] := tmp }

movej(lb,j,s,a)
int j,a[]
j >= lb, a[j] > s → { ; j := j − 1, movej(lb,j,s,a) }

movei(ub,i,s,a)
int i,a[]
i =< ub, a[i] =< s → { ; i := i + 1, movei(ub,i,s,a) }
```

Figure 2.1: Quick-Sort Program

in PCN: parallel composition of programs that operate on definitions and sequential composition of programs that operate on mutable state. The former of these views provides a mechanism for globally organizing parallel computation; the latter for achieving localized state changes. The interface between these views is provided by the equal and assignment operators.

2.1 Program Correctness

Correctness proof in PCN is based on proof rules for the three composition operators: sequential, parallel and choice. The proof rules for sequential composition are the same as those employed for normal sequentual languages e.g. Hoare triples [10]. We give proof rules for parallel composition under the assumption that *shared mutable variables remain constant during parallel composition*; these variables may be modified both before a parallel composition executes and after it terminates. Finally, as a consequence of the above assumption on parallel composition, choice composition is a simplified form of guarded commands [6] in which the guards are monotonic; provided that a guard is satisfied, progress is assured.

Using these concepts we now demonstrate a proof of correctness for the quick-sort program shown in Figure 2.1.

2.1.1 Correctness of the Partition Algorithm

We first define a predicate p as follows:

$$
\begin{array}{lll}
p = & (lb < ub) \wedge & (1) \\
& (lb \leq i \leq ub+1) \wedge & (2) \\
& (lb-1 \leq j \leq ub) \wedge & (3) \\
& s = a[lb] \wedge & (4) \\
& (\forall\, k : lb \leq k < i :: a[k] \leq s) \wedge & (5) \\
& (\forall\, k : j < k \leq ub :: a[k] > s) \wedge & (6) \\
& a[lb \ldots ub] \text{ is a permutation of } a'[lb \ldots ub], & (7) \\
& \quad \text{where } a' \text{ is the initial value of } a &
\end{array}
$$

Lemma. $p \Rightarrow (i \leq j+1)$

Proof. The implication follows from p.

$p \Rightarrow (j = ub) \vee (a[j+1] > s)$	from (6)
$(p \wedge (a[j+1] > s)) \Rightarrow i \leq j+1$	from (5)
$((j = ub) \wedge p) \Rightarrow i \leq j+1$	from (2)
$p \Rightarrow (i \leq j+1)$	from above three statements

Thrm. p is an invariant of partition.

Proof. p holds in the call to partition that occurs in qsort. $(lb < ub)$ due to the guard; (2),(3),(4) hold from the statements $s = a[lb], i := lb, j := ub$; (5),(6),(7) hold vacuously.

We shall prove that p holds after every block in the sequential composition of partition i.e. p holds after block_0, block_1 and block_2 in:

$$\{\ ?\ \mathsf{i} \leq \mathsf{j} \rightarrow \{\ ;\ \mathsf{block}_0,\ \mathsf{block}_1,\ \mathsf{block}_2\ \}\}$$

Lemma. movei maintains p. This occurs because: (1) and (4) are not modified; (2) and (5) follow directly from the code as invariants of movei; (3) and (6) are not modified. By the same argument, movej maintains p. Hence, block_0 maintains p.

- At the termination of movei, $(i > ub) \vee (a[i] > s)$.
- At the termination of movej, $(j < lb) \vee (a[j] \leq s)$.

At the termination of block_0:

$$((i > ub) \vee (a[i] > s)) \wedge ((j < lb) \vee (a[j] \leq s)) \qquad (8)$$

$$(p \wedge (i < j)) \Rightarrow (lb \leq i < j \leq ub)$$
$$(p \wedge (8) \wedge (i < j)) \Rightarrow (a[i] > s) \wedge (a[j] \leq s)$$

Hence:

$\mathsf{i} < \mathsf{j} \rightarrow$
Precondition: $(a[i] > s) \wedge (a[j] \leq s)$
swap(i,j,a),
Postcondition: $(a[j] > s) \wedge (a[i] \leq s)$, p holds.

Hence p holds at the recursive call to partition.

Proof of Progress. By induction.

Basis. Partition is correct for the case $i = j + 1$.

Proof. Invariant p holds, and substituting $j + 1$ for i gives the desired result:

$$(\forall k : lb \leq k \leq j :: a[k] \leq s) \wedge (\forall k : j < k \leq ub :: a[k] > s)$$

Inductive Step. Assume partition is correct for all i,j satisfying p and where $((j + 1 - i), v) \leq (m, w)$, for some integer m and where $(0 \leq m < (ub + 1 - lb)), w = 0, 1$ and $w = 0$ if $a[i] < a[j]$, 1 if $a[i] \geq a[j]$. We shall prove that partition is correct for all i,j satisfying p and where $(j + 1 - i) \leq m + 1$.

Proof. If $(j + 1 - i) > 0$ then either $(j + 1 - i)$ is reduced in the *move* block or w is reduced from 1 to 0. Thus, the metric reduces for successive calls to partition. Also p is an invariant of partition.

2.1.2 Correctness of the Quick-Sort Algorithm

The proof obligation for quicksort may be expressed as follows:

Precondition:

- lb, ub reduce to integers eventually
- a is an array of integers, $lb \geq 1$, $ub \leq maxa$, the dimension of array a

qsort(lb,ub,a)

Postcondition:

- lb, ub remain unchanged
- $a[lb \ldots ub]$ is a permutation of $a'[lb \ldots ub]$, where a' is the initial value of a
- $a[lb \ldots ub]$ is in ascending order, i.e., $(\forall k : lb \leq k < ub :: a[k] \leq a[k + 1])$

In addition, it is necessary to verify that shared mutable variables are constant during the parallel composition used in the algorithm. This involves the following proof obligation:

$$q \ : \ (\forall k : (k < lb) \vee (k > ub) :: a[k] \text{ is not accessed })$$

Proof by Induction. We define an invariant I of qsort, where:
$I = a[lb \ldots ub]$ is a permutation of $a'[lb \ldots ub]$. I holds initially since $a = a'$.

Base Case. $lb \geq ub$. In this case qsort(lb,ub,a) is a *skip*. The postcondition and q hold vaccuously.

Inductive Step. Assume *qsort* is correct for all $maxa \geq lb$, $ub \geq 1$, where $ub - lb \leq m$, for a nonnegative integer m, and prove qsort correct for all $maxa \geq lb$, $ub \geq 1$, where $ub - lb \leq m + 1 \leq maxa - 1$.

```
{ ? ub > lb →
        % ub > lb ∧ I
    { ;  s = a[lb], i := lb, j:= ub,
        % (ub > lb) ∧ (s = a[lb]) ∧ (i = lb) ∧ (j = ub) ∧ I
        % Hence I ∧ p
      partition(lb,ub,i,j,s,a)
        % I ∧ p ∧ (i = j + 1)
        % in particular:
        %   (∀k : lb ≤ k ≤ j :: a[k] ≤ s)∧
        %   (∀k : j < k ≤ ub :: a[k] > s)
      swap(lb,j,a)
        % I ∧ p ∧ r, where
        % r = (∀k : lb ≤ k < j :: a[k] ≤ s)∧
        % a[j] = s∧
        % (∀k : j < k ≤ ub :: a[k] > s)
        % (j − 1 − lb < (ub − lb)∧ from (2)
        % (ub − (j + 1)) < (ub − lb) from (3)

        % From the inductive assumption:
        % qsort(lb, j − 1, a) and qsort(j + 1, ub, a) are correct.
      { || qsort(lb,j-1,a), qsort(j+1,ub,a) }
        % I ∧ p ∧ r∧
        % a[lb . . . j − 1] is in ascending order ∧
        % a[j + 1 . . . ub] is in ascending order
        % Hence a[lb . . . ub] is in ascending order
```

In addition, it is necessary to verify the proof obligation for parallel processing, q is satisfied, using the same induction.

3 A Program Development Environment

In order to evaluate PCN on non-trivial applications, a core set of programming tools is required. These tools will be integrated into a program development environment during the initial phase of the project. The organization of the environment is shown in Figure 3.1.

All programming tools operate over a *uniform program representation* that allows them to interact. Each tool has a view of the representation that is implemented via a common set of *access functions.* The program representation includes dynamic information concerning run-time performance in addition to static information such as source code, correctness assertions, and performance models. The following core set of tools is required:

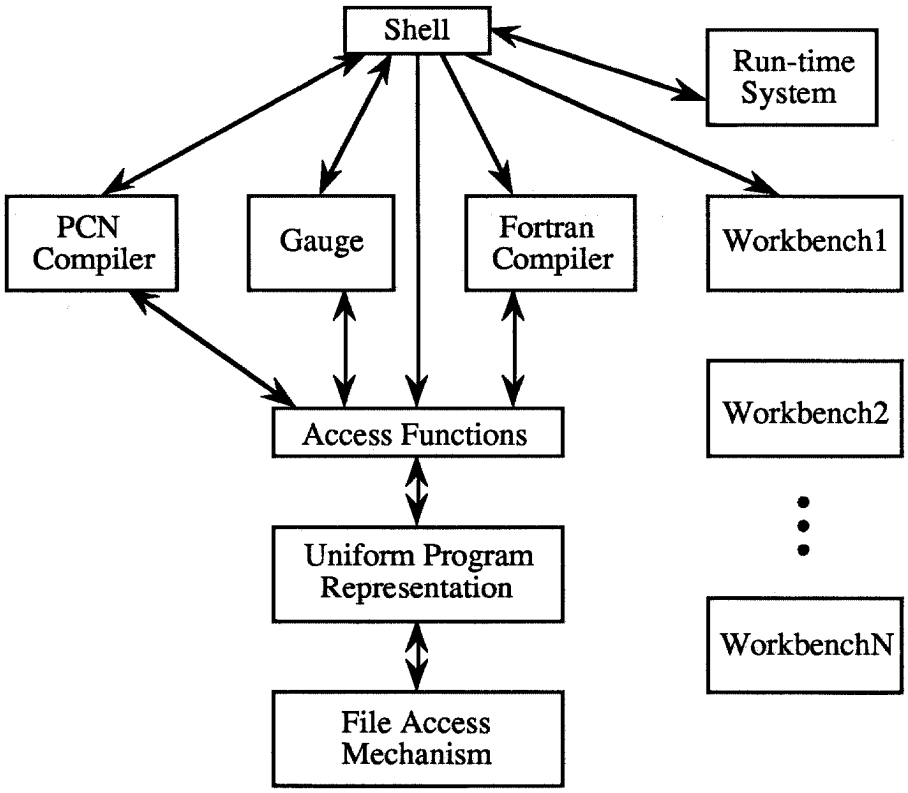


Figure 3.1: System Overview

Extensible Shell. A shell is a software device that provides applications and programmers with access to a variety of *capabilities*; these include system services (e.g. input/output, program mapping) and programming tools (e.g. compiler, debugger). A shell allows a programmer to interact with the system to begin the execution of one or more programs.

An extensible, interactive shell will be provided by the environment; it will allow programmers to customize the environment by dynamically adding or removing *capabilities*. The shell will be responsible for collecting performance data and depositing it into the uniform program representation.

PCN Compiler. The main task of the PCN compiler is to generate binary code that may execute using the run-time system; this code includes timers and counters that are maintained by the system. In addition, the compiler is responsible for building the static view of a program contained in the uniform program representation. This includes not only the program structure but also correctness assertions and a static performance model.

Run-Time System. The run-time system implements the abstract architecture described in Section 1.1 and is used by the shell to execute PCN programs. The in-

struction set of the abstract architecture supports communication and synchronization in PCN programs and forms the target for the PCN compiler. The run-time system maintains a collection of counters and timers that describe program execution, idle time, garbage collection, message volume, etc. This performance data can be retrieved by the shell and deposited into the uniform program representation for use by other tools.

Gauge. The performance evaluation tool, Gauge [15], provides a visualization tool that allows programmers to interactively explore a variety of performance metrics. These include static program characteristics, program timings, processor utilization, idle time and communication costs. The tool accesses the uniform program representation for both static and dynamic performance information. The static information consists of the performance models and program structure provided by the compiler. The dynamic information consists of counter and timer information maintained by the run-time system and provided via the shell.

Fortran Compiler. The optimizing Fortran compiler allows Fortran program segments to be compiled for the Intel i860; this machine is used in the Intel Delta multicomputer. Program components developed using this tool may be linked with the run-time system and called from within PCN programs. The tool may use correctness assertions via the uniform program representation for the purpose of optimization.

Additional Capabilities. The environment may be extended to include new *capabilities*, built in PCN, via the shell; these may include new programming tools, such as verifiers and debuggers, or *workbenches* for particular application domains. Section 5 describes how these extensions are achieved.

4 The Uniform Representation

The uniform program representation is in essence a simple database that provides a common medium for programming tools to interact [4]. For simplicity, it is specialized to support a single-user parallel programming environment; many common facilities associated with general databases (e.g., transaction scheduling, logging, etc.) are not supported.

The database representation is implemented using *data objects* that represent PCN programs, correctness assertions and performance data. All programming tools access the database via a common set of *access functions*; these separate the tools from the details of the representation and allow a degree of flexibility for modifications to the database structure. Figure 4.1 shows the overall structure of the database in terms of its constituent data objects; there are six data objects:

Module Node. A *module* is the unit of compilation and is represented by a module node. The primary data associated with a module node is the set of programs defined in the module and the call graph of these programs.

Program Node. For every PCN program there is one program node in the

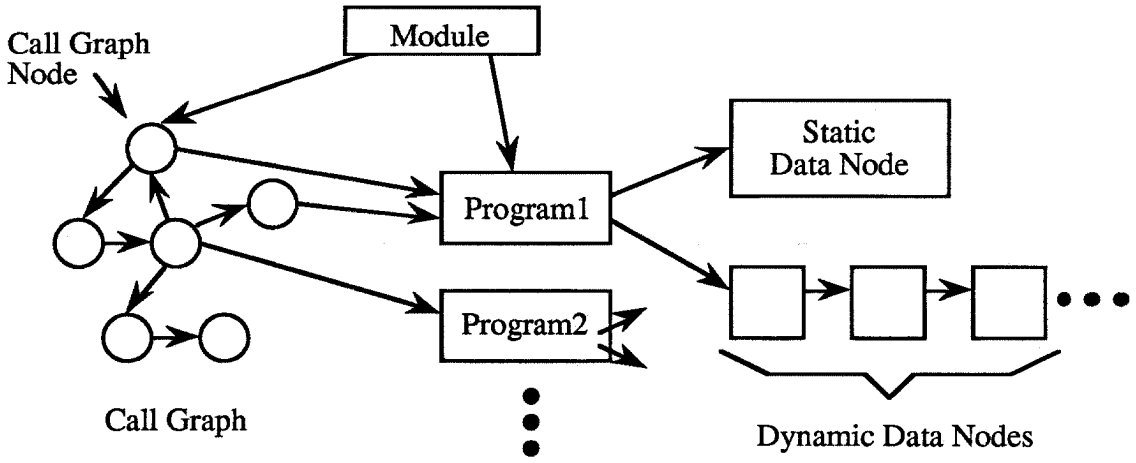


Figure 4.1: Database Structure

database. Associated with a program node are static and dynamic data nodes that contain information about the program and its execution.

Call Graph Node. Program nodes are linked together into a graph structure by call graph nodes. Each call graph node represents a call site in a program. Nodes in the graph can represent programs defined in other modules; these represent inter-module calls or calls to programs implemented in languages such as C or Fortran.

Static Data Node. The static data node stores static information about a program that does not vary between executions. This information includes performance models, the number of program arguments, and correctness assertions.

Dynamic Data Node. A set of dynamic data nodes is associated with each program. Each dynamic data node stores information concerning the execution of a program at a specific computing site. This information may be accessed using a variety of keyed access schemes.

Processor Node. A processor node contains architechtural parameters that characterize a unique class of computers. This structure is orthogonal to the main program representation and is not shown in Figure 4.1.

Manipulation of the data objects is provided by three access functions: *assert, retrieve* and *delete*. The assert function is used to add new information to the database. For example, the compiler may assert the existence of module and program nodes, while the performance tool may assert the existence dynamic performance data. The assert function can involve the creation of new nodes in the database and will maintain consistancy constraints in the database. The retrieve function provides keyed lookup of previously stored data. Information can be removed from the database using the delete function which also maintains consistancy constraints.

Although simple, the database is sufficiently general that it can represent a wide variety of program-partitioning strategies, mapping algorithms, heterogeneous com-

puting environments, and parallel architectures. It can easily be extended to support a variety of more complex structures, for example, *performance experiments* consisting of multiple executions of the same program or *snapshots* to support animations of program execution.

5 The Extensible Shell

Recall that a *shell* is a software device that provides applications and programmers with a variety of useful tools and system functions; these tools and functions are collectively termed *capabilities*. The programming environment provides a shell that executes at each computing site in a parallel machine; programmer interaction is handled at one particular site. The shell provides seven basic capabilities:

- **M:C.** This capability is used to initiate program execution; it causes a program in module M to be initiated with the call C.
- **profile(M:C,L).** The profile capability initiates program execution (i.e. M:C) and causes all modules named in list L to be profiled. The profile data is added to the uniform program representation when the program terminates.
- **load(M).** The load capability loads a code module M into the programming environment; programs defined in the module may subsequently be executed using the M:C capability. If the module defines new capabilities, they are added to the current shell capabilities; any existing capabilities are replaced.
- **unload(M).** The unload capability removes the code module M and any associated capabilities from the programming environment.
- **exit(C).** The exit capability allows the programmer to exit the environment. Termination occurs when the value of C is an integer; this value is used as a return code.
- **!S.** This capability allows the string S to be forwarded to the underlying UNIX system.
- **C@P.** This capability provides a mapping mechanism: It causes a capability C to be invoked at a position P within a parallel architecture. A variety of positional notations are available.

5.1 Defining New Capabilities

New capabilities may be defined by simply writing the appropriate PCN programs; these programs are then associated with a *server*. A server is a program that receives a stream of messages; it repeatedly removes a message from the stream and executes an

associated program. Programs may thus invoke capabilities by sending a message to the appropriate server. Servers may maintain state between invocations of capabilities and terminate only when their input stream closes.

Figure 5.1 implements a *compilation server* that provides two capabilities to the programming environment: compile and info. The compile capability causes a module to be compiled; the info capability simply returns the number of compilations that have been carried out.

The exports declaration (1) indicates that the module exports a single program: compile. The capabilities declaration (2) indicates the capabilities that the module provides to the environment: compile and info. Thus program compilation can be achieved either interactively from the programming environment or programmatically by another program. In contrast, information on the number of compilations performed can only be obtained from the environment. The server (3) maintains state corresponding to the the number of compilations invoked. This state is represented by the mutable integer Cnt which is initially set to zero (4). If a *compile* message is received (5), the state is updated (6) and compilation is performed (7). If an info message is received (9), the current state is returned (10) to the program that issued the request. In both cases, the server recursively consumes the remainder of the input stream (8,11). Compilation is achieved by invoking the various phases of the compiler which are defined in other modules (12); this is achieved using the M:C capability.

It is possible for the programmer to initialize the shell with a collection of default capabilities. This is achieved via a startup file (.pcnrc) that resides in the programmer's home directory. The programming environment begins execution by sending any statements in this file to the shell. To set up default capabilities the programmer simply loads modules that define the capabilities via this file; this mechanism provides the programmer with the ability to initialize the system with an appropriate *workbench.*

5.2 Core Servers

In order to provide the basic set of shell capabilities described at the beginning of Section 5, a collection of core servers must be available when the system is initialized. The following servers are required:

IO Server. This I/O server encapsulates input/output facilities and is responsible for handling the programmer's interaction with the run-time system (e.g., issuing prompts, etc.). If the shell is located at a computing site that does not have access to input/output devices, then the server ensures that information is routed to an appropriate site.

Code Server. The code server maintains a state corresponding to the binary code available at a computing site. The shell accesses this state for the purpose of initiating program execution (i.e. M:C).

Mapping Server. This server encapsulates streams to the other computing sites and implements the C@P capability. This capability provides the programmer with a

```
-exports(compile)                                   % exported routine (1)
-capabilities(compile,info)                         % environment capabilities (2)

server(Is)                                          % server of stream Is (3)
int Cnt
{ ;   Cnt := 0, serve(Is,Cnt) }                     % initialize state (4)

serve(Is,Cnt)
int Cnt
{ ?   Is ⇐ [compile(ModuleName)|Is1] →              % compile capability (5)
          { ;   Cnt := Cnt + 1,                     % change state (6)
                {||  compile(ModuleName),           % compile capability (7)
                     serve(Is1,Cnt)                 % serve rest of stream (8)
                }
          },
      Is ⇐ [info(Compiled)|Is1] →                   % info capability (9)
          { ;   Compiled = Cnt,                     % return current count (10)
                serve(Is1,Cnt)                      % serve rest of stream (11)
          }
}

compile(Name)                                       % compilation capability (12)
{||   tok:tokenize(Name,Tokens),                    % tokenize module
      par:parse(Tokens,Programs),                   % parse tokens
      pre:preprocess(Programs,FlatPCN),             % transform to Flat PCN
      en:encode(FlatPCN,Instructions),              % encode Flat PCN
      ass:assemble(Name,Instructions)               % generate binary
}
```

Figure 5.1: Compilation Capabilities

variety of *virtual machines* [12] that form convenient programming structures. Virtual machines are automatically mapped to each architecture by the programming environment. This allows application programs to be developed in an architecturally independent fashion and provides the ability to scale applications when additional computers become available.

Three virtual machines are available in the initial programming environment: a ring machine, a randomly addressed machine and an enumerated machine. The first allows instances of a program to be dynamically mapped around a ring using notations of the form p(...)@fwd or p(...)@bwd. The second provides the ability to randomly map programs with a uniform distribution; this uses a program notation p(...)@random. The last numbers the computing sites and allows program instances to be mapped by enumeration using a notation of the form {|| i in 1 .. N : p(...)@i }.

Server–Server. This server allows new servers to be added to the system and maintains a stream to each server for use by the shell.

Shell. The shell is itself a server: It receives a single stream of messages that emanate either from applications or from the programmer. The shell maintains a state corresponding to its capabilities and responds to requests to execute a capability. Input/output requests are forwarded to the I/O server. Mapping requests, of the form C@P, are forwarded to the mapping server. Any load and unload requests are handled by the I/O server; in addition, these requests may cause the shell to update its capability state and add new servers to the system via the server-server. The M:C and profile requests both initiate program execution and are handled by interaction with the code server; the profile request causes performance information to be added to the uniform representation when program execution terminates. The shell handles the exit and ! capabilities directly.

6 The PCN Compiler

The PCN compiler is separated into five distinct transformations that incrementally convert PCN source code into binary code that can execute under the run-time system. Figure 6.1 outlines these transformations.

The tokenizer is a finite state machine that transforms characters corresponding to program source text into elementary tokens. The parser transforms tokens into parse trees that represent program structure. A parse tree is a canonical program representation that can be used in a variety of source-to-source transformations; an overview of this representation is given in Appendix B. The preprocessor consists of a collection of source-to-source transformations, each of which implements a high-level PCN programming construct. The output of the pre-processor is a program that has only a core set of PCN constructs. The encoder transforms these core-PCN programs into a target set of abstract machine instructions; it also updates the uniform program representation to include information about the program being compiled. Finally, the

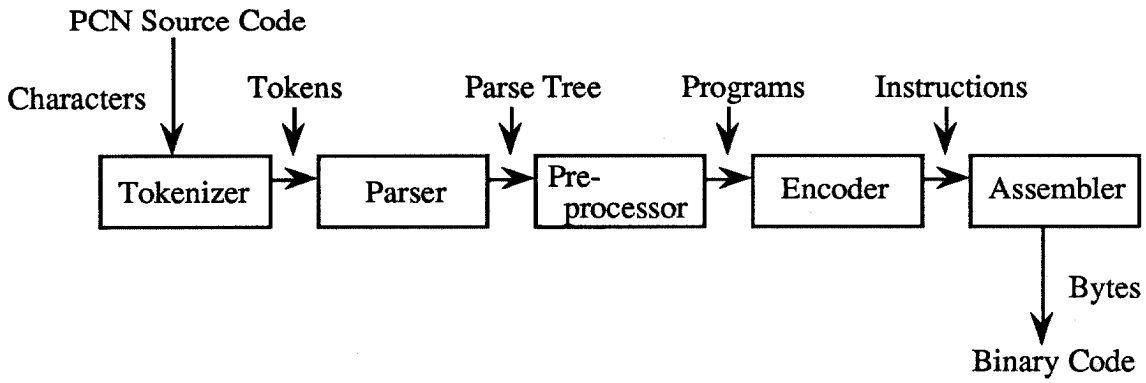


Figure 6.1: PCN Compiler Stages

assembler converts abstract machine instructions into a machine independent binary code format; this format includes counters and timers that are used for performance monitoring.

6.1 Preprocessing

Recall that the preprocessor takes a representation of a PCN program that includes high-level constructs and transforms it into a simple subset of the notation. This core notation has the following characteristics:

- It supports mutable variable declarations (e.g., `int x,y`), definitions (e.g., `X = Y`) and assignments (e.g., `X := Y`).
- It supports nested parallel composition.
- It only supports a single level of nesting in choice composition.
- It only supports a restricted form of sequential composition: This applies only to statements that can be ascertained, at compile time, to terminate immediately.
- It does *not* support quantification (e.g., `{ ; i in 1 .. N : p(i) }`).

This subset of the notation can be directly translated into executable statements. To support high-level PCN constructs such as a general form of sequential composition, nested-program composition or quantification, the compiler inserts additional program structures. Figure 6.2 shows the transformation of the partition algorithm that appeared in Figure 2.1 and outlines how the transformation operates.

Every PCN program is transformed into a simpler core-PCN program that has two additional arguments. These arguments represent a *synchronization chain* used to implement sequential composition. Tests are inserted into the new program to ensure

PCN Partition Code

```
partition(lb,ub,i,j,s,a)
int i,j,a[ ]
{ ?  i < j →
        { ;  {|| movej(lb,j,s,a), movei(ub,i,s,a) },          % A
             i < j → swap(i,j,a),                             % B
             partition(lb,ub,i,j,s,a)                         % C
        }
}
```

Core-PCN Code

```
partition(lb,ub,i,j,s,a,L,R)
int i,j,a[ ]
{ ? data(L), i < j →
     {||  movej(lb,j,s,a,L,M1),                               % A
          movei(ub,i,s,a,L,M2),
          barrier2(M1,M2,R1),
          new$1(i,j,a,R1,R2),                                 % B
          partition(lb,ub,i,j,s,a,R2,R)                       % C
     },
     default → R = [ ]
}

barrier2(M1,M2,R)
data(M1), data(M2) → R = [ ]

new$1(i,j,a,L,R)
int i,j,a[ ]
{ ?  data(L), i < j → swap(i,j,a,L,R1),
     default → R = [ ]
}
```

Figure 6.2: Transformation of the Partition Program

that the left of the chain is defined before the program executes; these data tests are shown in Figure 6.2. When a program terminates it defines the right of the chain (e.g. R:=[]). This mechanism implements the necessary sequencing operations required by sequential composition.

In parallel composition (A), the left of the chain is threaded to all the programs composed, this allows them to proceed concurrently; a barrier program is inserted to wait for all of the programs to terminate before allowing the right side of the chain to be defined.

Nested-choice composition (B) is implemented by inserting new programs; thus, programs are incrementally reduced to a single level of choice. The new$1 program inserted in Figure 6.2 demonstrates this transformation. This same technique is used to implement the PCN quantification construct; recursion is used to iterate over the quantified set.

6.2 Encoding

The encoding transformation takes core-PCN programs and generates abstract machine instructions that can be executed by the run-time system. An example of this encoding is provided in Section 7.2.

7 Run-Time System

The PCN run-time system implements an *abstract machine*; this machine corresponds to a single computing site in the abstract architecture described in Section 1.1. The abstract machine implements the PCN computational model; it represents the state of the computation and executes *abstract machine instructions* that encode tests on or modifications to the state. Programs to be executed on the abstract machine are encoded as sequences of these abstract machine instructions. The abstract machine may be implemented by an emulator written in a low-level language; alternatively, sequences of abstract machine instructions may be further compiled to machine code. The first approach will be adopted in the early stages of the project.

The PCN abstract machine design emphasizes simplicity and portability; no special support for sequential composition or nested choice blocks is provided. Hence, the machine can only execute directly programs expressed in the core-PCN subset described in Section 6.1. This approach is expected to provide reasonable overall performance, especially when programs include sequential components implemented in C or Fortran.

7.1 The Program Composition Machine

The Program Composition Machine (PCM) [9] is an abstract machine that supports the primitive operations in core-PCN programs; it comprises three components: *reduction*,

communication, and *garbage collection*. Each of these components executes at every computing site in a parallel architecture.

Reduction Component. This provides the facilities required to execute core-PCN programs. The reduction component maintains a pool of *processes* that correspond to program instances; it repeatedly selects a process and attempts to execute the associated program. This may cause additional processes to be placed in the pool or may cause the process to be returned to the pool because a needed definition is undefined; this latter situation can be viewed as *suspending* process execution until data is available.

Two important optimizations will be implemented to improve the efficiency of this basic model: support for *tail recursion* and a *scheduling structure*. Tail recursion permits execution to continue with a new process immediately when executing the body of a program. This avoids the overhead of adding the process to the process pool and subsequently selecting it. Tail recursion is only applied a finite number of times before the current process is added to the process pool and a new process is selected for execution. The number of tail recursive calls permitted before such a process switch occurs is termed the *time-slice*.

The scheduling structure avoids the overhead of repeatedly attempting to reduce processes that suspend. It consists of a single *active queue* containing all executable processes plus a *suspension structure* that links together processes that have suspended.

The abstract machine may read or write data when definitions are used in PCN programs. If this data resides at other computing sites, communication is employed; this effectively provides a global address space for definitions.

Communication Component. This operates at the end of a time-slice and receives messages. It can modify local data structures and/or send outgoing messages. Five types of message can be received: *Read*, *Define*, *Value*, *Cancel*, and *Collect*.

The *Read* message signifies that a remote site requests a copy of a definition when it is defined. The *Value* message carries a data structure and is received in response to a *Read* message. The *Define* message signifies that a remote site has executed a definition operation that refers to a local definition. The *Cancel* message indicates that a remote processor no longer requires intersite references. The *Collect* message signifies that a remote site requires garbage collection to be performed.

Garbage Collection Component. PCN supports automatic storage allocation and dynamic data structures; thus, a garbage collector is required to reclaim inaccessible storage. Global analysis techniques and program annotations can support optimizations that allow programs to execute in constant space; however, a garbage collector is required in the general case.

The garbage collector employed in the PCM has a global and a local component. The global component supports asynchronous garbage collection; that is, it permits individual sites to reclaim storage independently [8]. The local component employs a simple stop-and-copy algorithm [5].

7.2 The Instruction Set

The complete abstract machine instruction set is summarized in Appendix C and is described in detail in [9]. Figure 7.1 briefly illustrates the use of these instructions; it encodes a core-PCN program derived from the quick-sort program in Figure 2.1.

Core-PCN

```
movej(lb,j,s,a,L,R)
int lb,j,s,a[]
{ ?
        data(L), j >= lb, a[j] > s → { ; j := j - 1, movej(lb,j,s,a,L,R)},
        default → R = []
}
```

Abstract Machine Encoding

```
movej:  try(L1)                    % Start of 1st choice
            data(4)                % Ready?
            le(R0,R1)              % j ≥ lb
            build_data(R6,int,1)   % Create space for a[j]
            get_element(R1,R3,R6)  % Access a[j]
            lt(R2,R6)              % a[j] > s
            put_data(R7,1)         % Build integer 1
            sub(R1,R7,R1)          % j := j - 1
            recurse(movej,5)       % Recurse as movej
L1:     try(L2)                    % Start of 2nd choice
            default                % Succeed if 1st choice failed
            build_data(R6,tuple,0) % Create []
            define(R5,R6)          % R = []
            halt                   % Terminate process
L2:     suspend                    % Suspend process
```

Figure 7.1: Program Encoding

The encoding assumes that the programs arguments are loaded into abstract machine registers, beginning with register 0, when the process is scheduled. The try instructions encode the beginning of choices; their arguments are labels that indicate where execution should continue if the preconditions of an implication are not satisfied. Matching and test operations in guards are encoded using instructions such as le;

the build_data and get_element instructions are used to provide or access arguments. The body of an implication is encoded using instructions such as put_data, which creates a new integer, sub, which performs subtraction, and define, which encodes a definition. The recurse instruction encodes a tail-recursive call to a new program; halt encodes process termination. Another instruction, fork, is used for process creation.

7.3 Foreign Interface

Programs expressed in programming languages such as C and Fortran will be linked with the PCM. A call to one of these programs is compiled by the PCN compiler into a sequence of put_foreign instructions which set up a vector of pointers to arguments. These instructions are followed by a call_foreign instruction that invokes the foreign program. Further machine-dependent primitives are required to load foreign code dynamically and will not be provided in the initial implementation.

7.4 Performance Tools

The PCM incorporates low-level support for the Gauge profiling system. Each halt, recurse, and suspend instruction takes an offset to a counter as an argument and increments this counter each time it is executed. The call_foreign instruction takes an offset to a timer as an argument. This is used to accumulate the total time spent in the foreign program. The counters and timers are stored in code modules and can be accessed using the appropriate primitives.

Support is also provided for animation tools. A section of memory may be reserved for storing log information; a primitive allows programs to record events in this area. System facilities support the dumping of logged events.

8 Gauge

Current approaches to performance analysis of parallel programs rely on generating timestamped event logs. There are a number of limitations to this approach. The programmer needs *a priori* knowledge of program behavior to decide at what points in the program log entries are to be made. Instrumentation of the program thus requires either a good guess or performance data on the program. The techniques impose a high overhead in capturing performance data and programs often exhibit different behavior depending on whether data capture is turned on or off. The longer a program executes, the more performance data is collected. Instrumentation must be carefully placed to reduce performance data storage. Thus, performance has to be evaluated based on limited input data, or the instrumentation has to be changed as the input data set grows. Generating an accurate timestamp requires that the clocks on every processor be synchronized. The performance data is not available during program execution for runtime optimizations.

Rather than relying on logging, performance measurements in PCN will be accomplished by low-overhead instrumentation inserted into a program by the PCN compiler. Statistical models of program performance are generated by a static analysis of the program code. These models can be combined with values collected by the program-instrumentation and architecture-specific parameters to provide a number of performance indices. Experience indicates that performance estimates based on these indices are within 10% of their actual value. The runtime overhead for instrumentation is less than 3%; thus, we expect performance analysis to become an integral part of the program development cycle. Performance data will be stored in the uniform program representation; thus, different tools within the environment may access it.

Gauge will provide a visualization tool that promotes the interactive exploration of performance data. Performance metrics that will be available include execution time on a per-program basis, processor utilization, communication time and idle time. The tool will also provide interactive statistical analysis and data summary. An example processor-utilization display from this tool is shown in Figure 8.1; processors are listed horizontally while programs are listed vertically, the darkness of squares in the display indicated the level of computation.

Unlike existing performance display tools, we do not feel it is practical to view the activity of an individual processor as machines continue to scale. To solve this problem, data-classification algorithms will be used to compress and summarize data collected from sets of processors based on the type of tasks they perform.

9 Additional Tools

In addition to the core set of tools, the following tools are expected to be integrated into the system within a three-year time frame:

Template Interface. Program composition in PCN is based on a small number of methods. These *composition templates* can be displayed graphically using X-windows-based tools. In essence, by using a mouse and clicking on an icon, the programmer may summon an appropriate template. A program is constructed by *filling in the template blanks*; help and error checking can be provided interactively where necessary.

This simple interface would both speed program development and provide a PCN teaching aid for programmers unfamiliar with parallel programming concepts. The same tool may later form a basis for the interactive entry of program assertions to aid with debugging and proof checking.

Library Browser. Above we indicated that PCN programs may be constructed by filling in templates. The entries in these templates may refer to program libraries implemented in a variety of languages. For large-scale programs, an organizational tool will be constructed that will allow libraries to be browsed and used in template construction. The browser will assist in identifying and selecting components from libraries of existing code. For example, a request for *sorting programs* may result in

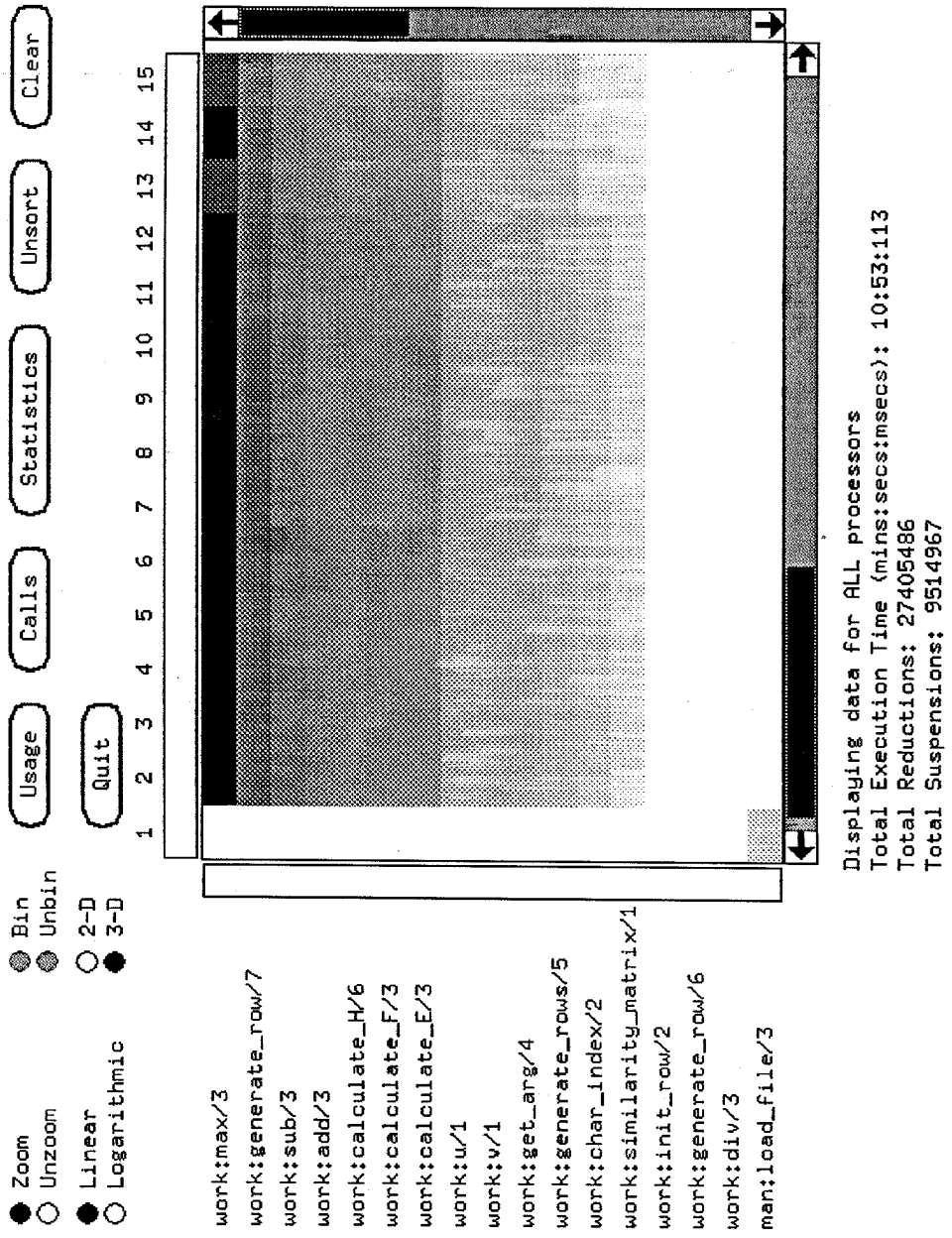


Figure 8.1: Example Gauge Display.

the identification of PCN programs for quick-sort, merge-sort, bubble-sort, etc. Help information associated with these abstractions will aid the programmer in selecting an appropriate program and adding it to a template.

Animation. The aim of PCN program animation [7] is to provide a tool that will allow programmers to gain insights into the operation of *critical components* of a computation. This can be achieved by capturing a small amount of data. This data characterizes the precise sequence of events that occur during program execution. The data can then be examined in post-mortem fashion using graphical displays.

The animation tool is intended to complement profiling tools such as Gauge. Gauge can be used to quantify the time spent executing various program components. This information permits a programmer to isolate critical procedures, detect load imbalances, etc. However, Gauge cannot be used to observe the fine details of program execution. Animation tools can provide this information and assist programmers in understanding and correcting performance anomalies identified by profiling.

Debugging Tools. A debugging tool based on formal methods will be provided. This tool will check program correctness assertions stated in PCN programs. In addition, a verification system, SDVS [11], will be evaluated to assess its applicability to PCN and its usefulness as a tool.

Fluid and Particle Flow Workbench. This workbench will provide composition operators specific to the application domain of fluid and particle flow simulation.

C and Ada. Program components expressed in C and Ada may be combined with Fortran components in a PCN program. Facilities to support this form of program composition will be provided using the available compilers.

10 Applications

To demonstrate the practical utility of the programming foundation a variety of application efforts will be conducted at the institutions involved in the project. The following sections briefly outline these application efforts.

10.1 Fluid Flow

The Applied Mathematics group at Caltech has invested considerable effort in the design of a large program concerned with the traveling-wave flow of incompressible viscous fluids. This program simulates Taylor vortices numerically by solving the discrete system of three-dimensional Navier-Stokes equations. The code has been carefully hand-optimized for efficient execution on Cray machines; machine time has proven to be prohibitively expensive. Scientists would like to obtain an alternative code that can execute on the less-expensive parallel machines available at Caltech.

From a computer science viewpoint, the code is *composed* of two distinct algorithms: a wave-front relaxation embedded within a multi-grid computation. Performance mea-

$$\forall i,j,k,t \quad (1 \le i \le I,\ \ 0 \le j \le J+1,\ \ 0 \le k \le K+1,\ \ 0 \le t \le T)$$

$$2 \le i \le I-1,\ \ 1 \le j \le J,\ 1 \le k \le K \ \rightarrow \qquad \% \text{ Interior Points}$$
$$v(i,j,k)^t \ = \ f1(v(i-1,j,k)^t,\ v(i,j-1,k)^t,\ v(i,j,k-1)^t,$$
$$v(i+1,j,k)^{t-1},\ v(i,j+1,k)^{t-1},\ v(i,j,k+1)^{t-1},$$
$$v(i,j,k)^{t-1})$$

$$1 \le j \le J,\ \ 1 \le k \le K \ \rightarrow \qquad \% \text{ i boundary}$$
$$v(1,j,k)^t \ = \ f2(v(2,j,k)^{t-1},\ v(3,j,k)^{t-1},\ v(1,j,k)^{t-1})$$
$$v(I,j,k)^t \ = \ f3(v(I-1,j,k)^t,\ v(I-2,j,k)^t,\ v(I,j,k)^{t-1})$$

$$1 \le i \le I,\ \ 1 \le j \le J \ \rightarrow \qquad \% \text{ k boundary}$$
$$v(i,j,0)^t \ = \ f4(v(i,j,K-1)^{t-1})$$
$$v(i,j,K+1)^t \ = \ f5(v(i,j,2)^t)$$

$$1 \le i \le I,\ \ 0 \le k \le K+1 \ \rightarrow \qquad \% \text{ j boundary}$$
$$v(i,0,k)^t \ = \ f6(v(i,J-1,k)^{t-1})$$
$$v(i,J+1,k)^t \ = \ f7(v(i,2,k)^t)$$

Figure 10.1: Parallel Relaxation Specification

surements on this code indicate that 70% of its running time is spent within the relaxation algorithm; thus, our preliminary investigations have focused on this aspect of the code. In essence, the relaxation algorithm implements a wavefront computation that moves through a torus and can be described abstractly by the parallel specification shown in Figure 10.1.

Notice how this algorithm describes a wavefront: For interior points in the torus the value at some point at time t is a function of:

- The value at that point at time $t-1$.
- Values at time t for neighbors behind the point.
- Values at time $t-1$ for neighbors in front of the point.

The boundary conditions effectively define the shape of the torus.

There are two particularly interesting aspects to this specification. The mathematics of the original code are embedded in the functions f1 through f7; these already exist in the sequential code and are implemented as large Fortran code segments. The parallel algorithm defined by this specification can easily be expressed in PCN and may reuse these large code segments. Secondly, the specification is an abstraction that forms an example of a programmer-defined composition. When parameterized, this

composition can be used to implement a wide variety of alternative relaxation schemes on parallel machines.

It is our hope that similar forms of composition occur in multigrid computations. By implementing the fluid flow code, we hope to uncover a set of abstractions of general usefulness to scientists working with numerical codes.

10.2 Trajectory Optimization

The trajectory optimization capability used at the Aerospace Corporation aids in the design of launch vehicle and spacecraft trajectories[14]. Given a mission orbit to which a spacecraft must be delivered, several parameters influence the paths that the launch vehicle and, subsequently, the spacecraft take enroute to achieving the orbit. These parameters might include the pitch profiles and jettison times for each of the lower stages, as well as the length, direction and inertial position of the spacecraft-orbit transfer burns. The optimization capability would be used to assist in selecting the combination of trajectory design parameters that would optimize some performance index. For example, a reasonable performance index might be to minimize the gross liftoff weight of the launch vehicle/spacecraft configuration.

Over many years, The Aerospace Corporation has refined a large computer program that will perform trajectory optimization. The current code is approximately 800,000 lines long and is written in a mixture of Fortran and assembly language. This code consumes a significant portion of the available computing resources at Aerospace. There is a known demand from the Air Force for an alternative to the current code that is both less costly and more responsive. In fact, the program is so expensive in terms of computer time that many program offices lack the funding that would enable them to use it. Other offices would like a more flexible capability that would allow changes in launch criteria closer to the launch date. Aerospace would like to accommodate the Air Force while at the same time releasing scarce computing resources.

Preliminary studies indicate that the core of the problem involves calculating multiple trajectories. The lack of data dependencies between these trajectories indicates a rich source of potential parallel execution. A 10,000-line subset of the larger code, which implements this aspect of the program, has been isolated. This subset forms the basis for a full-scale parallel implementation of the program.

10.3 Multi-Target Tracking

Multi-target tracking involves monitoring the path of multiple objects through space. Conceptually, this is similar to watching the radar screen of an air traffic controller: Multiple blips, or *returns*, are displayed that appear to move between sweeps of the radar. On successive sweeps, or *scans*, the returns that represent the same object in space must be associated based on a previous estimate of the object's speed and direction.

In the particular application that interests us, however, the incoming data is not from radar but from infra-red radiation such that no distance information can be derived using only one view of the object. Hence, two viewpoints are needed to stereoscopically fix and track the object through space. This requires that returns are associated not only from scan to scan but also between the two input sensors. If the field of vision of both sensors contains a large number of targets, then the problem of making associations and updating estimates of object velocity quickly becomes computationally intense.

The Aerospace Corporation has developed a large tracking program[13]. This program uses two input sensors; each produces a sequence of scans, a scan corresponds to a *snapshot* of the field of view. For each sensor, a window of three successive scans is processed to determine *feasible tracks.* This is achieved by computing the first and second differentials between the azimuth and elevation of returns on the scans. The feasible tracks from both sensors that correspond to one object must then be associated. This is achieved by scoring each pair of feasible tracks according to the distance of closest approach between two vectors. The vectors are drawn from each sensor to the objects in each sensor's field of view. An algorithm is then used to evaluate the scores and make the final associations. Using tracks that have been identified from previous scans, the current associations are used to update the position and velocity of existing tracks or to drop tracks that are no longer viable.

The tracker application offers many opportunities for exploiting parallel execution. Input from the two sensors can be buffered and formatted in parallel. The feasible tracks in each input stream can then be identified in parallel. For each pair of scans, the feasible tracks can be scored in parallel. Then, the track extensions can be computed in parallel. These are just a few of the possibilities for parallel execution that follow directly from the program's data structures: scans, returns and tracks. In addition, the tracking problem scales-up such that more processors can be used as the number of objects to be tracked increases.

10.4 Weather Modeling

Weather modeling is an interesting application area for PCN for several reasons. First, the computational requirements of global climate modeling necessitates parallel computing. Secondly, the technology currently used to implement climate models is primitive: Fortran on Crays. This technology hinders progress in model development because it is difficult to employ more modern numeric methods such as adaptive grids. (An adaptive grid employs dynamic grid refinement to focus on interesting regions such as fronts, thereby providing more accuracy for a constant cost.) We expect parallel implementations of adaptive systems to be easier in PCN since it supports recursively-defined data structures and automatic storage management. Adaptive grids also raise interesting load-balancing problems that are relatively easy to solve in PCN.

Argonne National Laboratory has already developed an initial parallel implemen-

tation of a weather-modeling code called MM4. This is a medium-scale (mesoscale) weather-modeling code developed at Penn State University and the National Center for Atmospheric Research (NCAR) in Boulder, Colorado. It solves a set of partial differential equations over a three-dimensional grid. In our experiments, we have used a 62x46x16 grid and a simulation time step of 160 seconds. Each grid square represents an 80x80 km square area; the atmosphere is divided into 16 layers.

The initial parallel implementation, along the lines we expect to use in PCN, employed a one-dimensional decomposition and gave a speedup of 10 on 16 Sequent Symmetry processors. We plan to develop a two-dimensional decomposition and expect to be able to exploit about 50 processors using 80x80 km resolution; significantly more at finer resolution. Meteorologists would like to go down to 5x5 km resolution.

In a second, related effort, we are assisting the Environmental Sciences Division at Argonne to construct a parallel implementation of the Regional Acid Deposition Model (RADM). This model uses data produced by MM4 to estimate acid deposition. Although RADM is smaller than MM4, it is more complex: 50 chemical processes are simulated at differing time scales. This introduces interesting load-balancing problems. In the first year of the project, we will produce PCN versions of both MM4 and RADM.

10.5 Computational Biology

A group at Argonne led by Ross Overbeek has been tackling problems in computational biology for two years [1]. Notable successes include the discovery of new secondary and tertiary structure in ribosomal RNA. A basic tool used extensively in this work is a program for automatically generating alignments of genetic sequences. The program that we currently use generates alignments similar to those produced manually by biologists.

Recently, we have designed a new algorithm that we expect to provide more precise alignments. We expect this algorithm to be suitable for execution on large multicomputers; existing sequential computers cannot cope with the thousands of sequences that biologists will soon need to align. The new algorithm generates an alignment between multiple sequences by estimating similarities between all pairs of sequences and using the similarities to construct a philogenetic tree (essentially, a minimum-cost-spanning tree). The algorithm then works up the philogenetic tree, combining pairs of sequences at each node. The alignment is complete when the root is reached. It should be clear from this brief description that the new algorithm offers considerable potential for parallel execution. It also includes complex data structures, potential sequential bottlenecks and interesting load-balancing problems. The design and implementation of an efficient parallel algorithm will provide a useful test of PCN in a non-numeric application. We will produce a PCN/C implementation of the new multiple-sequence alignment algorithm in the first year of the project.

References

[1] Butler, R. *et al., Aligning genetic sequences*, In book: Strand: New Concepts in Parallel Programming, Foster I. and Taylor, S., Prentice-Hall, 1989.

[2] Chandy, K.M. and Taylor, S., *Program Composition Notation*, To appear in book: Parallel Programming Languages, Springer, 1990

[3] Chandy, K.M. and Taylor, S., *The Composition of Parallel Programs*, To appear in: Proceedings of Supercomputing 89, Nov. 1989.

[4] Chyquanyou, R., Lee, C., Kessleman, C. and Taylor, S., *The Parallel Program Database Specification*, Tech. Rept., The Aerospace Corporation, In preparation.

[5] Cohen, J., *Garbage collection of linked data structures*, Computing Surveys, 13(3), 341-367, 1981.

[6] Dijkstra, E.W., *Guarded commands, nondeterminacy and formal derivation of programs*, CACM 18(8), pp 453-457, 1975.

[7] Disz, T. and Lusk, E., *A graphical tool for observing the behavior of parallel logic programs*, Proc. 1987 Symposium on Logic Programming, San Francisco, 1987, 46–53.

[8] Foster, I. *A multicomputer garbage collector for a single assignment language*, To appear in: International Journal of Parallel Programming, 1990.

[9] Foster, I. and Taylor, S., *A PCN Run-Time System*, Argonne National Laboratory, Tech. Rept. ANL/MCS-TM-137, 1990.

[10] Hoare, C.A.R. and Foley, M., *Proof of a recursive program*, Comp. Journal 14, pp 391-395, 1971.

[11] Redmond, T., *Composition of State Changes and Program Verification*, The Aerospace Corporation, Tech. Rept. ATR-86A(2778)-3 1987.

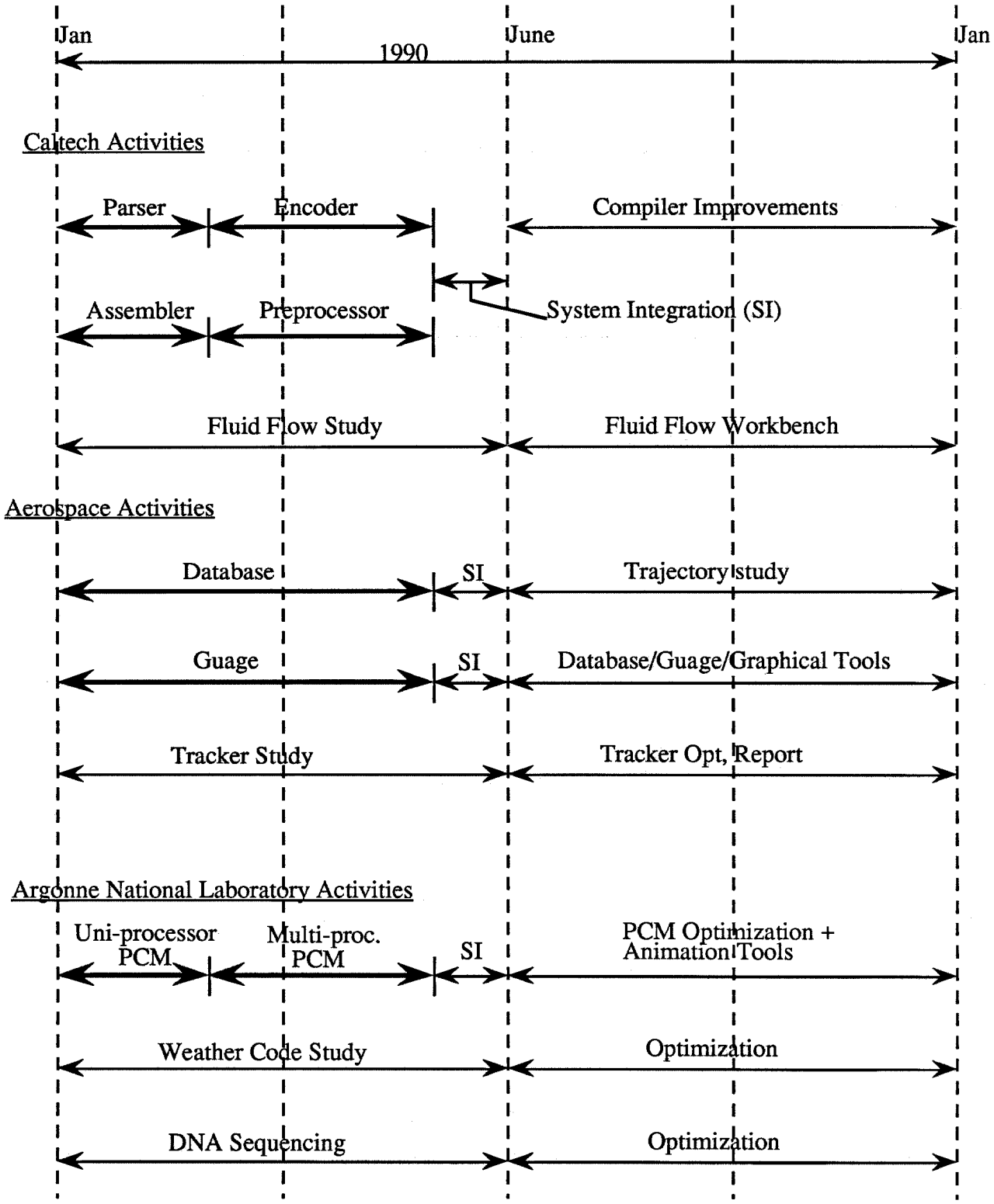
[12] Taylor, S., *Parallel Logic Programming Techniques*, Prentice-Hall, 1989.

[13] *Parallelization of a Tracking Algorithm*, The Aerospace Quarterly Technical Report, V8n2, April, 1989, p. 71–74.

[14] Trajectory Analysis Programming Department,*Generalized Trajectory Simulation Volume 1 : Overview*, The Aerospace Corporation, Tech. Rept. SAMSO-TR-75-255 Volume 1, dated 21 Nov 1975.

[15] Kesselman, C. and Gorlik, M., *Gauge: A workbench for performance analysis of logic programs*, International Conference on Logic Programming, 1988.

A Schedule of Activities



B Overview of the Parse Tree

The output of the parser is a parse tree that forms the internal representation of a PCN program; this representation forms the basis for all source-to-source transformations. The tree is constructed entirely of tuples and constants (i.e., numbers and strings); all lists structures (e.g., [X|Y]) are represented by binary trees built using tuples (e.g., {X,Y}). The central aspects of the representation are as follows:

- **Modules.** A module is represented by a list of programs; each program is represented by a tuple of the form {H,D,B}, where H represents the program heading, D the declarations and B the program body.
- **Headings and Calls.** A program heading or call of the form name(A1,A2,...An) is represented by a tuple of the form {'name',L} where L is a list of the representations for each argument.
- **Declarations.** A set of program declarations is represented as a list. A single declaration of the form type V is represented by a tuple of the form {'type', RV}, where RV is the representation for the variable V.
- **Variables and Definitions.** Variable, definitions and arrays are by a tuple of the form {'_var','String',L}, where String is the name of the variable and L is a list of subscripts or references.
- **Operators.** All binary infix operators are represented by a tuple of the form {'O',L,R}, where O is the operator; L is the left and R is the right operand. For example, A + B is represented by the tuple {'+',A,B}. Prefix and postfix operators are represented in a similar manner but do not have a second operand.
- **Expressions.** To distinguish the representation of an expression from an equivalent tuple, each expression is wrapped in a tuple of the form {'_exp',E}, where E is the representation of the expression constructed from its constituent operators.
- **Compositions.** Program compositions of the form {OP P1,P2,...Pn} are represented by a tuple of the form {'OP', L } where L is a list of the representations for programs P1,P2,...Pn.
- **Quantifications.** Quantified blocks of the form {OP Quantification : P1,...,Pn} are represented as a tuple of the form {'OP',Q,L} where Q represents the quantification and L is a list of the representations for blocks B1,...Bn. Quantifications take one of three forms: a list, variable or range of the form X..Y; the last form is represented as an infix operator.

Notice that this structure allows every construct in the notation to be uniquely detected by matching. This provides source-to-source transformations with a simple method for determining the structure of a program.

C Instruction Set and Assembler Output

Abstract Instruction	OP	B1	B2	B3	Word 2	Word 3
fork(Procedure,A)	0	A	Offset			
recurse(Procedure,A)	1	A	CntOff	Offset		
halt	2	0	CntOff			
suspend(A)	3	A	CntOff			
try(Label)	4	0	Offset			
run(R1,R2)	5	R1	R2	0		
send(R)	6	R	0	0		
build_data(R,B,T,Size)	7	R	B	T	Size	
build_tuple(R,B,Size)	8	R	B	0	Size	
build_def(R)	9	R	0	0		
put_data(R,B,S,Value)	10	R	B	S	Offset	Value1 ...
put_value(R)	11	R	0	0		
copy(R1,R2)	12	R1	R2	0		
get_tuple(R1,A,R2)	13	R1	A	R2		
equal(R1,R2)	14	R1	R2	0		
neq(R1,R2)	15	R1	R2	0		
type(R1,M,Tag)	16	R1	M	Tag		
default	17	0	0	0		
le(R1,R2)	18	R1	R2	0		
lt(R1,R2)	19	R1	R2	0		
data(R)	20	R	0	0		
sizeof(R1,R2)	21	R1	R2	0		
define(R1,R2)	22	R1	R2	0		
get_arg(R1,R2,R3)	23	R1	R2	R3		
get_element(R1,R2,R3)	24	R1	R2	R3		
put_element(R1,R2,R3)	25	R1	R2	R3		
add(R1,R2,R3)	26	R1	R2	R3		
sub(R1,R2,R3)	27	R1	R2	R3		
mul(R1,R2,R3)	28	R1	R2	R3		
div(R1,R2,R3)	29	R1	R2	R3		
copy_mut(R1,R2)	30	R1	R2	0		
put_foreign(R)	31	R	0	0		
call_foreign(Address)	32	0	TimeOff		Address	